Z3str4: A Multi-armed String Solver

Federico Mora[1](✉), Murphy Berzish[2], Mitja Kulczynski[3], Dirk Nowotka[3], and Vijay Ganesh[2]

[1] University of California, Berkeley, USA
fmora@cs.berkeley.edu
[2] University of Waterloo, Waterloo, Canada
[3] Kiel University, Kiel, Germany

Abstract. We present Z3str4, a new high-performance string SMT solver for a rich quantifier-free first-order theory of strings and length constraints. These kinds of constraints have found widespread application in analysis of string-intensive programs in general, and web applications in particular. Three key contributions underpin our solver: first, a novel length-abstraction algorithm that performs various string-length based abstractions and refinements along with a bit-vector backend; second, an arrangement-based solver with a bit-vector backend; third, an algorithm selection and constraint-sharing architecture which leverages the above-mentioned solvers along with the Z3 sequence (Z3seq) solver. We perform extensive empirical evaluation over 20 different industrial and randomly-generated benchmarks with over 120,000+ instances, and show that Z3str4 outperforms the previous best solvers, namely, CVC4, Z3seq, and Z3str3 in both total solved instances and total runtime.

Keywords: SMT · String solvers · Program analysis

1 Introduction

Security and reliability of string-intensive programs, especially web applications, is a significant problem that has received considerable attention in recent years by both industrial and academic researchers. A variety of analysis, testing, and verification methods have been proposed to address this problem [4,11,15,19,25,33,35,36,40], many of which depend on a string solver. As researchers continually improve their analysis tools and find new applications for string solvers, the demand for even more efficient solvers grows unabated.

To be used for these applications, string solvers must support a rich quantifier-free first-order theory T_S over string (a.k.a. word) equations, functions such as string concatenation and integer to string conversion, predicates such as string containment, and linear integer arithmetic over string length. Unfortunately, satisfiability problems for the theory T_S of strings (and its fragments) are generally hard. Specifically, the satisfiability problem for even the smallest interesting fragment of T_S, namely, the quantifier-free theory of word equations, is NP-hard and is in PSPACE [32]; the full theory T_S is undecidable [20]; and the question of decidability of the quantifier-free first-order theory of word equations and arithmetic over length remains open, despite many attempts to solve it over the last 50 years [31].

Despite their difficulty, much research has been done on practical algorithms for solving string constraints obtained from many real-world analysis, testing, verification, and synthesis applications [19,28,35,40]. Examples of such solvers include HAMPI [25], Stranger [42], Z3 sequence (Z3seq) [18], CVC4 [27], Norn [2], Trau [1], S3 [39], Z3str2 [43], and Z3str3 [9], each with varying strengths and weaknesses. Precisely because solving string formulas is believed to be hard in general, solver designers have come up with a diverse set of practical algorithms that incorporate a variety of tradeoffs. Some of these methods work well for pure word equations, but not so well for integer constraints over string length. Other methods work well for a mix of word equations and integer constraints, but perform poorly on more complicated constraints involving functions such as `substring` or predicates like `contains`. This diversity of algorithms presents an opportunity for effective *algorithm selection*.

Amadini [3] provides extensive documentation of this diversity, and classifies approaches to string constraint solving into three main categories: automata-based, word-based, and unfolding-based approaches. Automata-based approaches use finite automata to represent string variables and constraints. Word-based approaches reason about systems of word equations directly, while unfolding-based approaches expand string variables over the sequences of characters they represent. Solvers such as CVC4 [6], Z3str3 [9], and Z3seq [18] follow the word-based approach and implement a variety of algebraic proof rules, reductions, and axiomatizations for string formulas. Stranger [42] is an automata-based tool that uses reachability analysis together with finite automata representations of string constraints. The HAMPI tool [25] uses an unfolding-based approach, wherein the maximum length of each string variable is fixed *a priori* and solved using a reduction to bit-vector formulas.

We leverage this opportunity to apply algorithm selection for solvers, and introduce a new string solver, Z3str4, that incorporates two novel solver algorithms and several optimization methods aimed at algorithm selection. Z3str4 outperforms the previous best solvers, namely, CVC4, Z3seq, and the baseline Z3str3 overall on a diverse suite of 120,000+ industrial and randomly-generated instances from 20 different benchmarks (most of which were obtained from competing teams), on the basis of total number of instances solved and run time.

Contributions. We make the following contributions in this paper.

1. **A Length Abstraction Solver**. We present a length abstraction algorithm for solving string formulas based on abstractions and refinements of integer constraints. The algorithm iteratively refines an integer over-approximation of the input formula until it converges to the correct answer (See Subsect. 3.1).
2. **An Arrangement Solver with a Bit-Vector Backend.** We extend an existing arrangement method [9] for string solving via a conflict-driven clause learning-style abstraction-refinement string-to-bit-vector algorithm.

This hybrid approach combines the efficiency of an unfolding-based strategy with the ability of a word-based algorithm to reason about string terms of unbounded length (See Subsect. 3.1).

3. **Z3str4's Arm Selection Architecture.** We propose a variant of algorithm selection, *arm selection*, that selects sequences of algorithms to run on an input query. When one algorithm times out or gives up, the next algorithm is called immediately. Our arm selection method considers static and dynamic features. Additionally, algorithms within an arm can share constraints for additional performance benefits (See Subsect. 3.2).
4. **Extensive Experimental Results.** We combine the above-mentioned new methods as well as the Z3seq solver in an arm selection architecture, and call the resulting solver Z3STR4. We present an extensive experimental evaluation of Z3STR4 against three other state-of-the-art solvers, namely, CVC4, Z3seq, and Z3str3. Z3STR4 outperforms all these solvers overall. Results and code are available at `https://z3str4.github.io` (See Sect. 4).
   We also performed extensive evaluations against other well-known solvers such as S3, S3P, Norn, Trau, Stranger, and Ostrich. Unfortunately, all these solvers suffer from either significant soundness issues (Trau, S3, Norn, Ostrich) producing incorrect results, robustness issues, or crashes (S3P, Trau, Norn, Ostrich), or do not support relevant functions/predicates or Boolean operators that are part of the SMT-LIB 2.6 standard (e.g., Stranger does not support arbitrary string disequalities). Hence, we report on them only very briefly in Sect. 4.

2 Formal Background

In this section, we provide a brief overview of the input language accepted by Z3STR4 and the logical theory T_S considered in this paper.

2.1 Logical Theory T_S

For further details on the syntax and semantics of this theory, we refer the reader to the SMT-LIB standard [8].

Syntax. The Z3STR4 solver accepts input in the SMT-LIB format [8] following the current published standard for the theory of strings, and can handle quantifier-free formulas over Boolean combinations of string, integer, and regular expression (regex) formulas and terms. Atomic formulas handled by the string solver include string equalities and disequalities, regular expression membership, and extended string predicates such as `contains`, `prefixof`, `suffixof`, etc. Atomic formulas over integers, which may include inequalities, are handled by Z3's arithmetic solver. Boolean combinations of atomic formulas are handled by Z3's core solver in conjunction with the string and arithmetic solvers in a DPLL(T)-style approach. A summary of the basic syntax of the theory T_S is presented in Fig. 1. (We note that while we do support regex constraints, we shall not discuss them any further in this paper.)

$$\begin{array}{lll}
F & ::= & \mathit{Atom} \;\mid\; F \wedge F \;\mid\; F \vee F \;\mid\; \neg F \\
\mathit{Atom} & ::= & t_{str} = t_{str} \;\mid\; A_{int} \;\mid\; A_{ext} \;\mid\; A_{re} \\
A_{re} & ::= & t_{str} \in \mathit{RE} \\
A_{int} & ::= & t_{int} = t_{int} \;\mid\; t_{int} < t_{int} \\
A_{ext} & ::= & contains(t_{str}, t_{str}) \;\mid\; prefix(t_{str}, t_{str}) \;\mid\; suffix(t_{str}, t_{str}) \\
t_{int} & ::= & m \mid v \mid len(t_{str}) \mid t_{int} + t_{int} \mid m \cdot t_{int} \;\mid\; indexof(t_{str}, t_{str}, t_{int}) \;\mid \\
 & & str.to\_int(t_{str}) \text{ where } m \in Con_{int} \;\&\; v \in Var_{int} \\
t_{str} & ::= & s \;\mid\; v \;\mid\; t_{str} \cdot t_{str} \;\mid\; str.from\_int(t_{int}) \;\mid\; replace(t_{str}, t_{str}, t_{str}) \;\mid \\
 & & charAt(t_{str}, t_{int}) \;\mid\; substr(t_{str}, t_{int}, t_{int}) \text{ where } s \in Con_{str} \;\&\; v \in Var_{str}
\end{array}$$

Fig. 1. The syntax of the quantifier-free first-order theory T_S.

Semantics. String terms are composed of a finite (possibly empty) ordered sequence of characters taken from a finite alphabet, such as ASCII or Unicode. The expression $t_{str} \cdot t_{str}$ denotes string concatenation. For a string term w, $len(w)$ denotes the length of w as an integer number of characters. The empty string denoted by ϵ has a length of 0. Operations that refer to the index of a particular character or substring within another string use a zero-based index, that is, the first character of a string has an index of zero. The term `str.to_int` interprets a string as an integer by treating it as a non-negative number in base 10, possibly with leading zeroes. If the string represents a negative number or contains non-digit characters, the value is taken as -1. The term `str.from_int` converts a non-negative integer to the shortest possible string representing it in base 10. If the integer is negative, the value is taken as the empty string. Z3STR4 supports constraints over regular expressions, but we do not focus on regular expressions in this paper and instead refer the interested reader to [10].

The satisfiability problem for the quantifier-free theory T_S is to decide whether there exists an assignment of some constant in Con_{str} to every string variable in Var_{str} and some constant in Con_{int} to every integer variable in Var_{int} such that the formula evaluates to true. A formula is *satisfiable* (SAT) if such an assignment exists, and is *unsatisfiable* (UNSAT) if no such assignment exists.

3 Z3STR4 Components and Architecture

In this section, we describe the architecture of Z3STR4 and each of its components. Input to the Z3STR4 solver is given as an SMT-LIB formula, and the output is one of SAT, UNSAT, or UNKNOWN. Z3STR4 is built on top of Z3 and reuses its parser and core architecture. Once parsed, the formula is passed to Z3STR4's arm selection procedure, which makes use of a series of "probes" to analyze the formula and decide which of its arms is most appropriate for the given input. Each arm can call the novel length abstraction solver, the updated arrangement solver, and/or Z3's existing sequence solver in some predetermined order, as shown in Fig. 2. Z3STR4 moves to the next solver in the predetermined

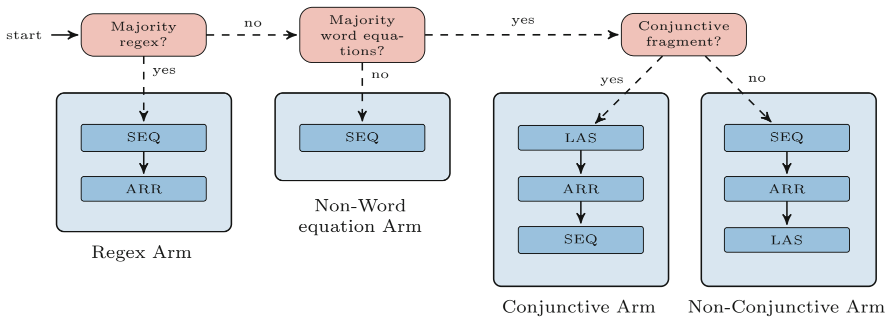

Fig. 2. Architecture of the Z3str4 tool. The red boxes indicate probes, and the blue boxes indicate algorithms. (Color figure online)

order when the current solver gives up. Solvers decide when to give up by using *dynamic difficulty estimation*, which we describe in Subsect. 3.2.

3.1 Novel Solver Algorithms in Z3str4

In this section, we describe the component algorithms of Z3STR4.

Z3str4's **Length Abstraction Solver.** We first describe a novel solving algorithm, called LAS, which uses an unfolding-based approach like HAMPI but overcomes the bounded length limitation by searching for length assignments. We begin with an overview of LAS's pseudocode in Algorithm 1, and then describe the subroutines it depends on in more detail. LAS takes in a conjunction ϕ of string literals, and returns either an assignment that satisfies ϕ or UNSAT. LAS begins by calling `MultisetCheck` at line 1. This subroutine can quickly determine UNSAT for many kinds of string constraints. If this check does not determine that the input is UNSAT, then LAS constructs θ_{lia}, an integer abstraction of ϕ, and enters its main solving loop. Every iteration of the loop updates θ_{lia}; the loop executes until either θ_{lia} is found to be UNSAT at line 5, or a satisfying string model is found at line 9. When θ_{lia} is SAT, we use a satisfying integer model, σ_{lia}, to reduce ϕ to a bit-vector query, θ_{bv}, and then check if θ_{bv} is SAT. If θ_{bv} is found to be SAT, at line 8, then we get a satisfying model, translate it to a satisfying string model, and return it as a solution. If θ_{bv} is found to be UNSAT, then we get an UNSAT core, update our length abstraction θ_{lia}, and repeat.

`MultisetCheck` is a heuristic that analyzes several static properties of atomic string formulas, and returns false if these formulas are UNSAT based on these properties. As an illustrative example, consider the word equation $0 \cdot X = X \cdot 1$. In order for this equation to be true, it is necessary that whatever value is assigned to X, the number of occurrences of each character on both sides must be equal. Since X appears on both sides exactly once, we can "cancel" it and

Algorithm 1: LAS Solver in Z3str4

```
Data: Conjunction of theory literals $\phi$
Result: Satisfying model or UNSAT
1  if $\neg$ MultisetCheck($\phi$) then
2      return UNSAT
3  end
4  $\theta_{\text{lia}} \leftarrow$ AbstractLengths($\phi$)
5  while LIACheck($\theta_{lia}$) $= SAT$ do
6      $\sigma_{\text{lia}} \leftarrow$ GetModel($\theta_{lia}$)
7      $\theta_{\text{bv}} \leftarrow$ ReduceToBV($\phi$, $\sigma_{lia}$)
8      if BVCheck($\theta_{bv}$) $= SAT$ then
9          return TranslateModel(GetModel($\theta_{bv}$))
10     else
11         $\gamma_{\text{lia}} \leftarrow$ Refine(GetUnsatCore($\theta_{bv}$), $\sigma_{lia}$)
12         $\theta_{\text{lia}} \leftarrow \theta_{\text{lia}} \wedge \gamma_{\text{lia}}$
13     end
14 end
15 return UNSAT
```

consider the remaining characters that appear in constant strings on each side. The left-hand side has a 0, but no 1s, and the right-hand side has a 1, but no 0s. From this we can conclude that the original equation must be false, since there is no way to assign X such that both sides have the same number of 0s or 1s. The `MultisetCheck` subroutine evaluates this heuristic by constructing the multisets of variables and characters that appear on each side of the equation and comparing them appropriately.

`AbstractLengths` takes the input query ϕ and returns an initial linear integer arithmetic length abstraction. The length abstraction contains simple length facts about the theory atoms in ϕ. For example, for the string equation $s = t$, we would assert $\text{len}(s) = \text{len}(t)$; for $\text{prefix}(s, t)$, we would assert $\text{len}(s) \leq \text{len}(t)$.

`ReduceToBV` takes a string formula f and an integer assignment σ (for string lengths and integer variables), and returns a bit-vector formula that is SAT iff $f \wedge \sigma$ is SAT. For example, given the formula $X = Y$, where X and Y are string variables, and the integer assignment $\sigma(\text{len}(X)) = 2 \wedge \sigma(\text{len}(Y)) = 2$, `ReduceToBV` would generate $X_1 = Y_1 \wedge X_2 = Y_2$, where all X_i and Y_i are 8-bit bit-vector variables. In this case, we say that the implied length constraint $\text{len}(X) = \text{len}(Y)$ *caused* the bit-vector equations $X_1 = Y_1$ and $X_2 = Y_2$. This bit-vector reduction is like that used by the HAMPI solver—we refer the reader to Ganesh et al. [25] for more details.

`LIACheck` and `BVCheck` take a linear integer arithmetic formula and bit-vector formula, respectively, and return either SAT or UNSAT. These two subroutines correspond to calling a linear integer arithmetic and bit-vector SMT solver, respectively. We implement this by calling a subsolver for the appropriate theory inside of Z3. For a concrete example, `BVCheck`$(X_1 = Y_1 \wedge X_2 = Y_2)$, where the input formula is as described above, would return SAT.

`GetModel` can only be used after a call to `LIACheck` or `BVCheck` that returned SAT. This subroutine returns a satisfying model from the previous solver. For example, calling `GetModel` after the call to `BVCheck` above, could return $\sigma_{\text{bv}}(X_1) = 11000001 \wedge \sigma_{\text{bv}}(X_2) = 11000010 \wedge \sigma_{\text{bv}}(Y_1) = 11000001 \wedge \sigma_{\text{bv}}(Y_2) = 11000010$. `TranslateModel` takes a bit-vector model and returns the corresponding string model. For example, given the bit-vector assignment σ_{bv}, `TranslateModel` would return $\sigma_{\text{s}}(X) = \text{“ab”} \wedge \sigma_{\text{s}}(Y) = \text{“ab”}$.

`GetUnsatCore` can only be used after a call to `LIACheck` or `BVCheck` that returned UNSAT. This subroutine returns a conjunction of theory literals that is a subset of the input, and is UNSAT.

`Refine` takes the UNSAT core of a bit-vector reduction θ_{bv} and the integer model θ_{lia} that produced θ_{bv}, and returns a length constraint that will at least block the length assignment θ_{lia}. In many cases, `Refine` does better and blocks more than one assignment. This is formalized in Theorem 1. To get an intuition for how refine works, we first walk through an example. Consider the input query $X \cdot X \cdot Y = Y \cdot 1 \cdot 2 \cdot X \wedge Y = X \cdot 3$ with the length assignments $\text{len}(X) = 2$ and $\text{len}(Y) = 3$. In this case θ_{bv} would be

$$X_1 = Y_1 \wedge X_2 = Y_2 \wedge X_1 = Y_3 \wedge X_2 = 1 \wedge Y_1 = 2 \wedge Y_2 = X_1 \wedge Y_3 = X_2 \\ \wedge Y_1 = X_1 \wedge Y_2 = X_2 \wedge Y_3 = 3$$

where X_i and Y_i are characters at position i of X and Y, respectively. An UNSAT core for θ_{bv} would be $X_2 = 1 \wedge Y_3 = X_2 \wedge Y_3 = 3$. Given this UNSAT core, `Refine` will generate the constraint $\text{len}(X \cdot X) \neq \text{len}(Y \cdot 1) \vee \text{len}(X \cdot X \cdot Y) \neq \text{len}(Y \cdot 1 \cdot 2 \cdot X) \vee \text{len}(Y) \neq \text{len}(X \cdot 3)$, which simplifies to $2 \cdot \text{len}(X) \neq \text{len}(Y)+1$ when taking into account the initial length abstraction. In other words, `Refine` creates a constraint that ensures that the same pairs of characters in an UNSAT core are never aligned again. These lessons are general when the bit-vector queries do not contain disjunctions and when the counterexamples fall on concatenation boundaries. If `Refine` is unable to learn a general lesson, as described in Theorem 1, it will negate θ_{lia}.

Theorem 1 (Simple, General Refine). *Let ϕ_{bv} be the bit-vector reduction of a word equation $X^1 \cdot \ldots \cdot X^n = Y^1 \cdot \ldots \cdot Y^m$ for some length assignment. Suppose $C := X_i = Y_j \wedge \ldots \wedge X_l = Y_k$ is a bit-vector UNSAT core for ϕ_{bv}, where every X_a and Y_b are the characters at positions $len(X^1 \cdot \ldots \cdot X^a)$ and $len(Y^1 \cdot \ldots \cdot Y^b)$ of the word equation, respectively, and let C_{len} be C but with every bit-vector equality replaced with the corresponding length constraint that caused each equality in ϕ_{bv}:*

$$C_{len} := len(X^1 \cdot \ldots \cdot X^i) = len(Y^1 \cdot \ldots \cdot Y^j) \wedge \ldots \wedge len(X^1 \cdot \ldots \cdot X^l) = len(Y^1 \cdot \ldots \cdot Y^k).$$

Then any lengths that satisfy C_{len} will cause an UNSAT bit-vector reduction.

Proof. Construct the same UNSAT core up to renaming of bit-vector variables. □

The main loop of Algorithm 1 returns UNSAT when there are no integer solutions to explore, and it returns SAT if a satisfying bit-vector model is found.

At a high-level, this algorithm is correct because `Refine` never blocks length assignments that could lead to a satisfying bit-vector reduction, and because every bit-vector model corresponds to a string model. Unfortunately, Algorithm 1 is not guaranteed to terminate.

Z3str4's Arrangement Solver with a Bit-vector Backend. The arrangement solver invoked by Z3str4 is a variant of the one implemented in the Z3str3 solver [9]. The key modification is the introduction of a bit-vector backend which is invoked after arrangement reduction has finished. The bit-vector backend reuses many of the components of LAS, but it follows the conflict-driven clause learning architecture of Z3str2 [43] and Z3str3.

Solving Strings via Arrangements. We briefly describe the arrangement solver for completeness, and refer the reader to previous papers [9,43] on this topic for a more thorough description. The core idea behind the arrangement technique is the reduction of string equations to simpler string equations until the formula is in "solved form": a conjunction of equations of the form $X = t$, where X is a string variable that appears in the input formula, t is a concatenation of string constants and possibly fresh string variables, and every variable in the input formula appears exactly once. See Section 4.3 of Ganesh et al. [21] for the complete, original definition of solved form. When considering an equality between strings, the arrangement technique introduces a disjunction of formulas describing the possible relationships between variables on the left-hand side and right-hand side of the equation. For example, for the equality $A \cdot B = X \cdot Y$, there are three possible relationships between A and X: either A and X have the same length, or A is shorter than X, or A is longer than X. The arrangement technique expresses the possible relationships with the following three implied formulas: either $A = X$ and $B = Y$, or $A \cdot X_1 = X$ and $B = X_1 \cdot Y$ for a fresh string variable X_1, or $A = X \cdot X_2$ and $X_2 \cdot B = Y$ for a fresh string variable X_2. These three formulas are asserted as a disjunction to the core solver, which chooses one branch to explore. The solver continues to reduce and explore the resulting formulas until either (1) it has reduced the entire formula to solved form or (2) it encounters an *overlap*. We describe the use of the novel bit-vector backend for both cases and define overlaps in the next two paragraphs.

`ReduceToBV` for Solved Form. The arrangement solver makes use of the same subroutines for reduction of strings to bit-vectors (`ReduceToBV`, `BVCheck`, `GetModel`, and `TranslateModel`) described for LAS. The bit-vector reduction is invoked after arrangement reduction completes and the integer constraints have been solved by Z3's arithmetic theory solver. That is, the bit-vector backend is used as the model construction algorithm once the solver has found a set of formulas in solved form. If the bit-vector reduction returns UNSAT, we block the corresponding assignment in Z3's core solver and instruct it to continue the search with a different length assignment or a different arrangement. This repeats until Z3str4 converges to the correct answer and returns SAT, or the core solver finds a top-level conflict and returns UNSAT. Note that unlike LAS,

$$\begin{array}{lll}
F & ::= & Atom \mid F \wedge F \mid \neg G \mid A_{int} \vee A_{int} \mid \neg A_{int} \\
G & ::= & G \vee G \mid \neg F \\
Atom & ::= & t_{str} = t_{str} \mid A_{int} \mid A_{ext} \\
A_{int} & ::= & t_{int} = t_{int} \mid t_{int} < t_{int} \\
A_{ext} & ::= & prefix(t_{str}, t_{str}) \mid suffix(t_{str}, t_{str}) \\
t_{int} & ::= & m \mid v \mid len(t_{str}) \mid t_{int} + t_{int} \mid m \cdot t_{int} \\
 & & \quad \text{where } m \in Con_{int} \text{ \& } v \in Var_{int} \\
t_{str} & ::= & s \mid v \mid t_{str} \cdot t_{str} \mid charAt(t_{str}, t_{int}) \\
 & & \quad \text{where } s \in Con_{str} \text{ \& } v \in Var_{str}
\end{array}$$

Fig. 3. Context-free grammar for conjunctive fragment.

which invokes the arithmetic solver as a procedure, the arrangement solver uses the string and integer theory integration approach described previously [43].

`ReduceToBV` **for Overlaps.** An important weakness of Z3str3's arrangement solver is that it cannot handle word equations which have the same variable occurring on both the left hand and right hand side of an equation, referred to as an overlapping variable. Consider the equation $0 \cdot X = X \cdot 0$. Z3str3's solver would detect the existence of an overlapping variable and return UNKNOWN. However, Z3str4 easily handles such equations. Observe that once string variable lengths have been fixed, string equations can still be reduced to bit-vectors even if they contain overlapping variables. For example, again considering $0 \cdot X = X \cdot 0$, if the arithmetic solver proposes the candidate model $len(X) = 2$, the bit-vector reduction would reduce X to the 8-bit bit-vector characters $x_1 x_2$ and solve the bit-vector equation $0x_1x_2 = x_1x_20$, finding it satisfiable with solution $X = 00$.

3.2 Algorithm Selection and Clause Sharing

We now describe how the component algorithms of Z3str4 are combined using the arm selection procedure and the clause-sharing mechanism. The arm selection method uses static features of the instance to determine which of the three solver algorithms to invoke and in what order. Dynamic difficulty estimation determines when to move to the next solver in order.

Static Arm Selection. The input formula ϕ is first passed to Z3's simplifier and term rewriting procedure. The algorithm selection procedure then follows a three-tiered sequence of checks for static features, illustrated in Fig. 2. The order and choice of solvers to use was determined by a combination of empirical results and experimentation. First, if any regex constraints appear in the input formula ϕ, the arrangement solver is used. Otherwise, if a majority of top-level formulas in the input are *not* word equations, the sequence solver is used. Finally, the algorithm selection procedure calls the `ConjunctiveFragment` subroutine. This subroutine returns TRUE if the query is in the language generated by the grammar in Fig. 3, and FALSE otherwise. We call this language the conjunctive fragment. When a query is in the conjunctive fragment, we call LAS first, followed by the arrangement solver and the sequence solver. When a query is not in the conjunctive fragment, we call the sequence solver first, followed by the arrangement solver and the LAS solver.

The conjunctive fragment is effective because queries in the language are guaranteed to reduce to conjunctions of bit-vector equations when lengths are fixed (formalized in Theorem 2). This means that every iteration of LAS is quick (bit-vector solvers are efficient in this fragment), and LAS is more likely to learn general lessons (see Theorem 1).

Theorem 2 (Conjunctive Fragment). *Let L be the language generated by the grammar in Fig. 3. If $\varphi \in L$ is an input query, then* LAS *will always call* `ReduceToBV` *such that it produces a conjunction of bit-vector equations.*

Proof. LAS receives conjunctions of theory literals from the core solver (ϕ from Algorithm 1 is a solution to the Boolean abstraction of the input query φ). Show by induction on the grammar L that equality, prefix and suffix predicates must always appear under an even number of negations. Finally, show that theory literals that use these predicates will reduce to a conjunction of bit-vector equations iff they appear under an even number of negations. $\square$

Dynamic Difficulty Estimation. Rather than giving solvers a fixed time budget, we give up on them when it is unlikely that they will terminate. We call this process of monitoring the internal state of a solver and determining when to give up *dynamic difficulty estimation*. We incorporated dynamic difficulty estimation into the Z3 sequence solver and LAS.

At a high level, the Z3 sequence solver works as a sequence of checks: if a check fails, then a corresponding action is taken (for example, asserting an implied formula) and the process repeats; if all checks pass, then the query is solved. In total, the sequence solver has 20 of these checks. We observe that queries that are "easy" for the sequence solver rarely fail later checks. Our difficulty estimation monitor, therefore, keeps track of the current latest failing check, and we give up when the latest failing check is the i^{th} check, where i can be specified as a solver parameter. Empirically, we find the best check to give up on to be the second to last check, `branch_nqs`. Additionally, we found that monitoring the number of automata propagations and calls to `solve_eq` performs similarly well.

LAS generates a bit-vector query that is solved by an external bit-vector solver. We measure two main aspects of LAS's state: the number of times `ReduceToBV` has been called, and the amount of time taken by each bit-vector check. We find that the benefit of each successive iteration diminishes rapidly, and that the time taken by the bit-vector solver tends to increase with each successive call (due to finding larger integer models). We fix the maximum number of iterations and the maximum time budget for a bit-vector check. When the solver exceeds either limit, we move to the next solver. Empirically, LAS solves most queries in the conjunctive fragment in under five iterations.

Constraint Sharing in Z3str4. When Z3str4 switches from one solver to the next, it is useful to propagate information. This ensures that subsequent solvers will not get "stuck" exploring the same search space. We implemented a mechanism in Z3's SMT architecture wherein a theory solver can indicate during the search that one or more constraints are to be shared to future solvers in the event that it fails. Specifically, the LAS solver and arrangement solver both share blocked length assignments learned during the search. The only requirement for this to remain sound and complete is that each shared constraint must be implied by the original input formula and cannot contain any new variables that do not appear in the original formula.

4 Performance Evaluation

In this section, we report on the performance of Z3STR4 and how it compares against state-of-art solvers, CVC4, Z3seq, and Z3str3, over 20 different benchmark suites obtained from industrial applications as well as solver developers.

Description of Benchmark Suites. We evaluated Z3STR4 and competing solvers over 20 benchmark suites containing 120366 instances covering a wide range of applications. This is, to the best of our knowledge, the largest and most comprehensive collection of instances known for testing string solvers. Most of these suites were curated by solver developers or industrial users.

Of the 20 suites we used in our experimental evaluation, 16 are from various published sources, while one (BanditFuzz) is previously unpublished. The following benchmark suites come from well-known applications: Kaluza [35], PyEx [34], SMTLIB25 [7], PISA [37], IBM AppScan [43], Leetcode Strings [1], Sloth [22], Cashew [13], JOACO [38], Kausler [24]. The following suites were published by solver developers: Z3str3 Regression [9], Norn [2], Woorpje Word Equations [16], Trau Light [1], Stranger [42], StringFuzz [12], Automatark [10], stringfuzz-regex-generated [10], and stringfuzz-regex-transformed [10].

Competing String Solvers. We compared Z3STR4 against three leading string solvers, namely CVC4 [6], Z3str3 [9], and Z3seq [18].[1] We used the following criteria in determining which solvers to compare against: all of them had to be reasonably efficient on a diverse set of benchmarks, support the entirety of the string theory as standardized by the SMT-LIB initiative, be robust (have zero or very few crashes), and be reliable (have zero or very few wrong answers).

We also performed detailed evaluation of many other solvers, including S3, S3P [39], Norn [2], Trau, Z3-Trau [1], Ostrich [14], and Stranger [42]. Unfortunately, these solvers suffer from either significant soundness issues (S3, Norn, Trau, Ostrich), robustness issues (Trau, S3, Norn, Ostrich), or do not support string operations used in the benchmarks (Norn does not support the entire SMT-LIB input language; Ostrich does not support string length; Stranger does not support arbitrary string disequalities). Hence, we had to exclude them from our experimental evaluation. For example, Z3-Trau had more than 3,500 soundness errors and over 500 segmentation faults on our benchmarks.

[1] We used the binary version of CVC4 1.8 from cvc4.github.io. For Z3str3 and Z3seq, we used the commit `#7e7360dd0c04cdee95c3f74a59908209742c5212` of the official repository of the Z3 solver (github.com/Z3Prover/z3).

Table 1. Cumulative results. Timeout = 20 s. Total time includes all solved, timeout, unknown and error instances. "Virtual Best Z3str4" represents perfect selection between Z3seq, LAS, and our novel arrangement solver. "Virtual Best Overall" represents perfect selection between CVC4, Z3seq, Z3str3, and Z3str4.

	CVC4	Z3seq	Z3str3	Z3str4	Virtual Best Z3str4	Virtual best overall
sat	68386	69853	59663	**71842**	74012	74096
unsat	43897	43783	41198	**44597**	44659	44923
unknown	**40**	50	949	170	454	113
timeout	8043	6680	18556	**3757**	1241	1234
soundness error	**0**	**0**	9	**0**	0	0
program crashes	**0**	**0**	**0**	**0**	0	0
Total correct	112283	113636	100852	**116439**	118671	119019
Time (s)	187941.021	176134.056	391379.159	**107456.964**	50996.414	44527.707
Time w/o timeouts (s)	27080.148	42534.056	**20259.159**	32316.964	23843.007	19289.888

Test Environment. All experiments were performed on a server running Ubuntu 18.04.4 LTS with two AMD EPYC 7742 processors and 2TB RAM using the ZaligVinder [26] benchmarking framework. We extended ZaligVinder with the ability to cross-validate models produced for SAT instances between solvers. The timeout for solving an instance was set at 20 s.

4.1 Overall Evaluation

Table 1 shows a summary of the results for all 20 benchmark sets, and is read as follows: the line "(un)sat" counts the number of instances classified (un)satisfiable, "Unknown" counts the number of instances where a solver returned UNKNOWN, and "Timeout" counts the number of instances where a solver exceeded its wall-time limit of 20 s. We also include the number of errors and crashes observed for each solver. "Total correct" counts the number of correctly classified instances. "Time" measures the total wall time to solve all instances, including timeouts, unknowns, and errors. The last line shows the total time excluding all timeouts. The rightmost columns of Table 1 present the "virtual best solvers:" hypothetical tools that always run the fastest solver for every query from an input set of solvers. The first virtual best solver, "Virtual Best Z3str4," selects between the algorithms within Z3STR4's portfolio (Z3seq, LAS, and our novel arrangement solver). The second virtual best solver, "Virtual Best Overall," depicts perfect selection among the state-of-the-art solvers (CVC4, Z3seq, Z3str3, and Z3str4). The same data is visually depicted as a cactus plot in Fig. 4. A detailed view containing the cumulative results for each benchmark group is available at https://z3str4.github.io.

Overall, Z3STR4 outperforms CVC4, Z3str3, and Z3seq, solving more instances and having a lower total solving time than every other solver and with no errors or crashes. Z3STR4 solves 4156 more cases than CVC4, 2803 more cases than Z3seq, and 15587 more cases than Z3str3. Including timeouts,

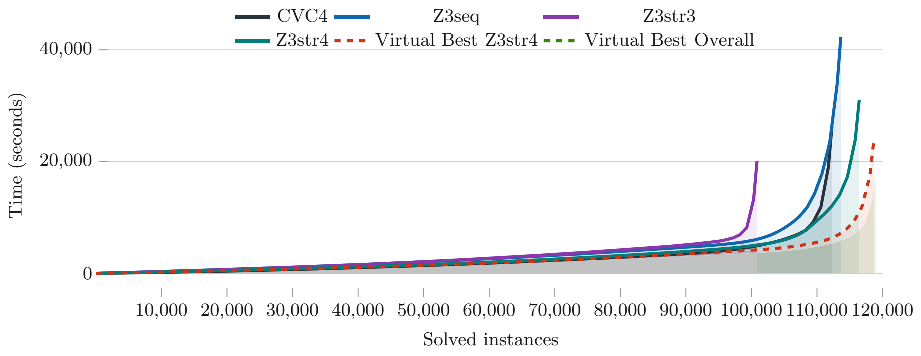

Fig. 4. Cactus plot of string solvers on all benchmarks. Timeout = 20 s. Timeout, unknown, and error instances excluded.

Z3STR4 is 75% faster than CVC4, 64% faster than Z3seq, and 264% faster than Z3str3. Z3STR4 approaches the overall virtual best solver in terms of number of queries solved (97.83%) and comes the closest in terms of total time taken (141%). Notably, our algorithm selection strategy approaches the optimal selection strategy for our portfolio of solvers ("Virtual Best Z3str4") and solves only 2232 fewer instances. The overall results indicate that Z3STR4 is highly effective at solving a wide variety of practical string instances.

4.2 Performance Analysis of Components of Z3STR4

To evaluate our architecture and to better understand our component algorithms, we categorize the queries by the arm they are assigned to and compare our component algorithms on the queries they are meant to do well on versus the queries they are not meant to do well on. There are 35345 regex queries, 42522 higher-order queries, 15113 conjunctive queries, and 23643 non-conjunctive queries. We exclude 3743 queries that are solved by the simplifier.

LAS **Performance Analysis.** We hypothesize that LAS will do comparatively better in the conjunctive fragment because it will learn more general lessons and its underlying bit-vector solver will be quicker every iteration. Empirically this hypothesis holds: LAS solves more queries per second compared to the arrangement solver in the conjunctive fragment (1128.5%) than it does outside the conjunctive fragment (904.6%). These high percentages are largely due to dynamic difficulty estimation, which lets LAS return unknown before wasting too much time on a query. Overall, we find that LAS is extremely effective in the conjunctive fragment, especially when used as the first solver in an arm.

Arrangement Solver Performance Analysis. The arrangement solver with our novel bit-vector backend significantly improves performance over the arrangement solver without the bit-vector backend, both in terms of time and number of instances solved. Without the bit-vector backend, the arrangement solver solves 72417 instances in 423949.163 s; with the bit-vector backend, the arrangement solver solves 107401 instances (148.3% of the queries without) in 262047.893 s (61.8% of the time without).

Sequence Solver Performance Analysis. Empirically, we find that the sequence solver is best when most constraints are not word equations. In this fragment, the sequence solver solves 39639 cases in 74565.085 s, while the next best solver, the arrangement solver, solves only 31376 (79.1%) in 220076.933 s (295.1% of the sequence solver's time).

Impact of Clause Sharing. Clause sharing significantly reduces the amount of time taken and slightly increases the number of solved instances. In particular, over all benchmarks, with clause sharing turned off, Z3STR4 solves 15 fewer cases and takes 2201.790 more seconds (102% of the time taken with clause sharing).

5 Related Work

Theory and Practice of String Solvers. Makanin showed in 1977 that the theory of quantifier-free word equations was decidable [30]. Plandowski later showed that this problem was in PSPACE [32]. Continuing the aforementioned thread of research, Ganesh et al. proved that satisfiability for quantified word equations with a single quantifier alternation is undecidable, as well as that satisfiability of the quantifier-free SMT-LIB theory of strings, including string-integer conversion, is also undecidable [20,21]. Many extensions to the theory of word equations have been shown to be undecidable [17,20,23,29,30,32].

HAMPI [25] is one of the first string solvers that used the idea of reducing fixed-length string constraints to bit-vectors. While highly effective, HAMPI does not support unbounded string variables, as many newer tools now do. Z3STR4's arrangement and length abstraction solvers use a similar reduction to bit-vectors, but do so in an abstraction refinement fashion that enables them to support constraints with arbitrary-length string variables. An interesting detail about Z3STR4's method for fixing string variable lengths is that it uses Z3's arithmetic solver to obtain length assignments for input string constraints that are consistent with input length constraints.

The Z3 theorem prover [18] is a DPLL(T)-based SMT solver for theory combinations over first-order logic. Z3 includes an arithmetic solver for linear integer arithmetic, and a sequence solver that supports word-based reasoning over strings. Z3STR4 uses Z3's sequence (Z3seq) solver as part of both arms considered during its algorithm selection. The Z3str3 solver [9] is based on Z3 and the previous Z3str2 solver [43]. It uses a reduction known as the arrangement technique to convert word equations into simpler formulas until a so-called "solved form" is reached. Z3STR4 uses a version of Z3str3 that has been extended with a string-to-bit-vector reduction, better heuristics for handling formulas with "overlapping variables" which Z3str3 (and Z3str2) have difficulties dealing with, and the ability to share certain learned clauses between invocations of the solver. The CVC4 solver [6] handles constraints over the theory of strings and arithmetic using an algebraic approach, and uses a similar DPLL(T) architecture to Z3. Norn [2] is an automata-based solver that solves integer arithmetic constraints using finite automata and then represents word equations with finite automata that have been restricted with respect to concrete length constraints. Stranger [42] is another automata-based approach, but based on a static analysis technique that determines possible solutions of a string variable while traversing an automaton. Ostrich [14] implements another technique using transducers to solving string constraints. A stand-alone solver named Trau [1] uses an approach which looks for simple patterns inside the input formula within a CEGAR framework.

Algorithm Selection for Solvers. SATZilla [41] is a portfolio-based algorithm selection strategy for Boolean SAT problems that uses many features of a SAT formula to predict the performance of each SAT solver in its portfolio and decide which solver to use for a given instance. SATZilla uses machine learning to train a predictive model of solver performance. FastSMT [5] uses machine learning to choose Z3 probes and tactics, a form of fine-grained method selection. After training on queries from one domain, the FastSMT generated strategy can be used to speed up the performance of Z3 on subsequent queries from that domain. FastSMT has been successfully applied to formulas which include bitvectors, integer arithmetic, and real arithmetic. We differ from FastSMT in that they are limited to existing probes, while we designed our own, and we also use dynamic information in addition to static features.

6 Conclusion and Future Work

We presented a new string solver, Z3str4, which supports the entirety of the SMT-LIB standard for strings. Z3str4 includes two novel algorithms for solving string constraints: a length abstraction algorithm and an arrangement solver with a string-to-bit-vector backend. Both of these algorithms use a variety of abstractions and refinements combined with a bit-vector reduction. We also describe an arm selection architecture which uses static features of an instance and dynamic information about solver state to get the best out of each algorithm. We demonstrated the performance of Z3str4 over a comprehensive evaluation of 20 industrial and randomly-generated benchmarks and over 120,000 individual instances, showing that Z3str4 outperforms leading string solvers.

As a future extension to this work, we plan to revisit the method by which algorithm selection is performed. Currently, the algorithm selection architecture is limited to choosing between fixed "arms" which have been hard-coded. Machine learning may give us the opportunity to learn a more sophisticated function for algorithm selection. Furthermore, we plan to improve the way in which constraints are shared between different algorithms within an arm by exploring the tradeoffs between constraint size and number of shared constraints.

Acknowledgments. We thank the anonymous reviewers whose comments and suggestions greatly improved our paper. This work was supported in part by NSF grants CNS-1739816 and CCF-1837132, by the DARPA LOGiCS project under contract FA8750-20-C-0156, by the iCyPhy center, and by gifts from Intel, Amazon, and Microsoft.

References

1. Abdulla, P.A., et al.: TRAU: SMT solver for string constraints. In: 2018 Formal Methods in Computer Aided Design (FMCAD), pp. 1–5. IEEE (2018)
2. Abdulla, P.A., et al.: Norn: an SMT solver for string constraints. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 462–469. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_29
3. Amadini, R.: A survey on string constraint solving (2020)
4. Backes, J., et al.: Semantic-based automated reasoning for AWS access policies using SMT. In: Bjørner, N., Gurfinkel, A. (eds.) 2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, 30 October–2 November 2018, pp. 1–9. IEEE (2018). https://doi.org/10.23919/FMCAD.2018.8602994
5. Balunovic, M., Bielik, P., Vechev, M.: Learning to solve SMT formulas. In: Bengio, S., Wallach, H., Larochelle, H., Grauman, K., Cesa-Bianchi, N., Garnett, R. (eds.) Advances in Neural Information Processing Systems, vol. 31, pp. 10337–10348. Curran Associates, Inc. (2018). http://papers.nips.cc/paper/8233-learning-to-solve-smt-formulas.pdf
6. Barrett, C., et al.: CVC4. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 171–177. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_14
7. Barrett, C., Fontaine, P., Niemetz, A., Preiner, M., Schurr, H.J.: SMT-LIB benchmarks. https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks. commit 11f52315
8. Barrett, C., Fontaine, P., Tinelli, C.: The Satisfiability Modulo Theories Library (SMT-LIB). www.SMT-LIB.org (2016)
9. Berzish, M., Ganesh, V., Zheng, Y.: Z3str3: a string solver with theory-aware heuristics. In: 2017 Formal Methods in Computer Aided Design (FMCAD), pp. 55–59. IEEE (2017)
10. Berzish, M., et al.: An SMT solver for regular expressions and linear arithmetic over string length. In: Silva, A., Leino, K.R.M. (eds.) CAV 2021. LNCS, vol. 12760, pp. 289–312. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-81688-9_14
11. Bjørner, N., Tillmann, N., Voronkov, A.: Path feasibility analysis for string-manipulating programs. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 307–321. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-00768-2_27
12. Blotsky, D., Mora, F., Berzish, M., Zheng, Y., Kabir, I., Ganesh, V.: StringFuzz: a Fuzzer for string solvers. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10982, pp. 45–51. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96142-2_6
13. Brennan, T., Tsiskaridze, N., Rosner, N., Aydin, A., Bultan, T.: Constraint normalization and parameterized caching for quantitative program analysis. In: Bodden, E., Schäfer, W., van Deursen, A., Zisman, A. (eds.) Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, 4–8 September 2017, pp. 535–546. ACM (2017). https://doi.org/10.1145/3106237.3106303

14. Chen, T., Hague, M., Lin, A.W., Rümmer, P., Wu, Z.: Decision procedures for path feasibility of string-manipulating programs with complex operations. In: Proceedings of the ACM on Programming Languages, vol. 3, no. POPL, pp. 1–30 (2019)
15. D'Antoni, L., Veanes, M.: The power of symbolic automata and transducers. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017. LNCS, vol. 10426, pp. 47–67. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63387-9_3
16. Day, J.D., Ehlers, T., Kulczynski, M., Manea, F., Nowotka, D., Poulsen, D.B.: On solving word equations using SAT. In: Filiot, E., Jungers, R., Potapov, I. (eds.) RP 2019. LNCS, vol. 11674, pp. 93–106. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-30806-3_8
17. Day, J.D., Ganesh, V., He, P., Manea, F., Nowotka, D.: The satisfiability of word equations: decidable and undecidable theories. In: Potapov, I., Reynier, P.-A. (eds.) RP 2018. LNCS, vol. 11123, pp. 15–29. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-00250-3_2
18. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
19. Emmi, M., Majumdar, R., Sen, K.: Dynamic test input generation for database applications. In: ISSTA, pp. 151–162 (2007)
20. Ganesh, V., Berzish, M.: Undecidability of a theory of strings, linear arithmetic over length, and string-number conversion. CoRR arXiv:1605.09442 (2016). http://arxiv.org/abs/1605.09442
21. Ganesh, V., Minnes, M., Solar-Lezama, A., Rinard, M.: Word equations with length constraints: what's decidable? In: Biere, A., Nahir, A., Vos, T. (eds.) HVC 2012. LNCS, vol. 7857, pp. 209–226. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39611-3_21
22. Holík, L., Janku, P., Lin, A.W., Rümmer, P., Vojnar, T.: String constraints with concatenation and transducers solved efficiently. In: PACMPL, vol. 2, no. POPL, pp. 4:1–4:32 (2018). https://doi.org/10.1145/3158092
23. Jez, A.: Recompression: a simple and powerful technique for word equations. In: Proceedings of STACS, LIPIcs, vol. 20, pp. 233–244 (2013)
24. Kausler, S., Sherman, E.: Evaluation of string constraint solvers in the context of symbolic execution. In: Proceedings of ASE - IEEE/ACM, pp. 259–270. ACM (2014)
25. Kiezun, A., Ganesh, V., Guo, P.J., Hooimeijer, P., Ernst, M.D.: HAMPI: a solver for string constraints. In: Proceedings of the Eighteenth International Symposium on Software Testing and Analysis, ISSTA 2009, pp. 105–116 (2009). https://doi.org/10.1145/1572272.1572286
26. Kulczynski, M., Manea, F., Nowotka, D., Poulsen, D.B.: The power of string solving: simplicity of comparison. In: Proceedings of AST (2020)
27. Liang, T., Reynolds, A., Tinelli, C., Barrett, C., Deters, M.: A DPLL(T) theory solver for a theory of strings and regular expressions. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 646–662. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_43
28. Lin, A.W., Majumdar, R.: Quadratic word equations with length constraints, counter systems, and Presburger arithmetic with divisibility. In: Lahiri, S.K., Wang, C. (eds.) ATVA 2018. LNCS, vol. 11138, pp. 352–369. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-01090-4_21

29. Lin, A.W., Barceló, P.: String solving with word equations and transducers: towards a logic for analysing mutation XSS. In: Bodík, R., Majumdar, R. (eds.) Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, 20–22 January 2016, pp. 123–136. ACM (2016). https://doi.org/10.1145/2837614.2837641
30. Makanin, G.: The problem of solvability of equations in a free semigroup. Math. Sbornik **103**, 147–236 (1977). English transl. in Math USSR Sbornik 32 (1977)
31. Matiyasevich, Y.: The connection between Hilbert's tenth problem and systems of equations between words and lengths. Semin. Math., V. A. Steklov Math. Inst., Leningrad **8**, 61–67 (1968). translation from Zap. Nauchn. Semin. Leningr. Otd. Mat. Inst. Steklov 8, 132–144 (1968)
32. Plandowski, W.: An efficient algorithm for solving word equations. In: Proceedings of the 38th Annual ACM Symposium on Theory of Computing STOC 2006, pp. 467–476 (2006). https://doi.org/10.1145/1132516.1132584
33. Redelinghuys, G., Visser, W., Geldenhuys, J.: Symbolic execution of programs with strings. In: Proceedings of the South African Institute for Computer Scientists and Information Technologists Conference, SAICSIT 2012, pp. 139–148 (2012). https://doi.org/10.1145/2389836.2389853
34. Reynolds, A., Woo, M., Barrett, C., Brumley, D., Liang, T., Tinelli, C.: Scaling Up DPLL(T) string solvers using context-dependent simplification. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017. LNCS, vol. 10427, pp. 453–474. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63390-9_24
35. Saxena, P., Akhawe, D., Hanna, S., Mao, F., McCamant, S., Song, D.: A symbolic execution framework for JavaScript. In: Proceedings of the 2010 IEEE Symposium on Security and Privacy, SP 2010, pp. 513–528. IEEE Computer Society, Washington (2010). https://doi.org/10.1109/SP.2010.38
36. Sen, K., Kalasapur, S., Brutch, T., Gibbs, S.: Jalangi: a selective record-replay and dynamic analysis framework for JavaScript. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2013, pp. 488–498. ACM, New York (2013). https://doi.org/10.1145/2491411.2491447
37. Tateishi, T., Pistoia, M., Tripp, O.: Path- and index-sensitive string analysis based on monadic second-order logic. ACM Trans. Softw. Eng. Methodol. **22**(4), 33:1–33:33 (2013). https://doi.org/10.1145/2522920.2522926
38. Thomé, J., Shar, L.K., Bianculli, D., Briand, L.: An integrated approach for effective injection vulnerability analysis of web applications through security slicing and hybrid constraint solving. IEEE TSE **46**(2), 163–195 (2018)
39. Trinh, M.T., Chu, D.H., Jaffar, J.: S3: a symbolic string solver for vulnerability detection in web applications. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, pp. 1232–1243 (2014)
40. Wassermann, G., Su, Z.: Sound and precise analysis of web applications for injection vulnerabilities. In: Ferrante, J., McKinley, K. (eds.) PLDI, pp. 32–41. ACM (2007)
41. Xu, L., Hutter, F., Hoos, H.H., Leyton-Brown, K.: SATzilla: portfolio-based algorithm selection for SAT. CoRR arXiv:1111.2249 (2011)
42. Yu, F., Alkhalaf, M., Bultan, T.: Stranger: an automata-based string analysis tool for PHP. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 154–157. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-12002-2_13
43. Zheng, Y., et al.: Z3str2: an efficient solver for strings, regular expressions, and length constraints. Formal Meth. Syst. Des. **50**(2–3), 249–288 (2017)